Wright *et al. Journal of Cloud Computing: Advances, Systems and Applications* 2012, **1**:6
http://www.journalofcloudcomputing.com/content/1/1/6

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

A constraints-based resource discovery model for multi-provider cloud environments

Peter Wright, Yih Leong Sun*, Terence Harmer, Anthony Keenan, Alan Stewart and Ronald Perrott

Abstract

Increasingly infrastructure providers are supplying the cloud marketplace with storage and on-demand compute resources to host cloud applications. From an application user's point of view, it is desirable to identify the most appropriate set of available resources on which to execute an application. Resource choice can be complex and may involve comparing available hardware specifications, operating systems, value-added services (such as network configuration or data replication) and operating costs (such as hosting cost and data throughput). Providers' cost models often change and new commodity cost models (such as spot pricing) can offer significant savings. In this paper, a software abstraction layer is used to discover the most appropriate infrastructure resources for a given application, by applying a two-phase constraints-based approach to a multi-provider cloud environment. In the first phase, a set of possible infrastructure resources is identified for the application. In the second phase, a suitable heuristic is used to select the most appropriate resources from the initial set. For some applications a cost-based heuristic may be most appropriate; for others a performance-based heuristic may be of greater relevance. A financial services application and a high performance computing application are used to illustrate the execution of the proposed resource discovery mechanism. The experimental results show that the proposed model can dynamically select appropriate resouces for an application's requirements.

Introduction

Infrastructure providers offer flexible and cost-effective resources for hosting network-centric and cloud applications. An infrastructure provider rents compute and storage resources together with network bandwidth and supporting services, according to prespecified user requirements, for precisely the length of time that a user requires them. Rented resources may be used to (i) host all of an application's infrastructure, or (ii) support overflow capabilities during high-load situations, or (iii) provide disaster recovery capabilities. The cost of renting infrastructure resources is inexpensive due to economies of scale. Moreover, an infrastructure can be tuned to the current load of an application or the current revenue generated by an application.

Infrastructure providers are increasingly supplying the cloud marketplace with storage and on-demand compute resources to host cloud applications. Amazon Elastic Compute Cloud (EC2) [1], ElasticHosts [2], GoGrid [3], Flexiscale [4] and Rackspace [5] all supply resources to the *IaaS* (Infrastructure as a Service) market. Each infrastructure provider offers a particular infrastructure capacity, with a variety of hardware configurations, operating systems and supporting services. Different providers offer different pricing structures for using their infrastructure resources and they may have different application programming interfaces (APIs) for requesting and configuring resources. This makes it difficult for users to migrate between providers within a multi-provider cloud marketplace in order to minimize the cost of using resources.

One way to utilise the cloud marketplace is to develop models for mapping application constraints onto ranges of infrastructure products. As the infrastructure provider marketplace develops and user expectations increase, providers are introducing a richer set of pricing models and value-added services. For example, Amazon now offers *Spot Prices* for their resources; these allow resources to be obtained at significant discounts to their normal fixed price structures. Other providers offer ranges of network bandwidth options and services, such as resilience and load scaling. These capabilities are added dynamically. In order to automate the process of resource selection for

*Correspondence: ysun05@qub.ac.uk
Belfast e-Science Group, Queen's University Belfast, Belfast, UK

a particular application, techniques for expressing infrastructure and software requirements are needed. From the application user's point of view, there is a challenge to find appropriate resources within a multi-provider cloud. Searching for infrastructure resources by hardware specifications, such as CPU or memory, may not be sufficient. A user may be interested in searching for resources by operating system type or by requiring particular software items or services. For example, a user may require two compute nodes with hardware configurations of 2.4GHz dual core Intel CPU, 2GB memory, 250GB local disk, running a 32-bit Ubuntu 9.10 Karmic system pre-installed with JDK 1.6 and Tomcat 5.0 web server, with each compute node being located at a different geographical location (for resilience) with a low network latency. A user should be able to compare easily alternative options and select resources from different infrastructure providers.

In this paper we propose and build an infrastructure resource discovery engine which operates in a multi-provider cloud marketplace with multiple kinds of user requirements (including cost). In particular we utilise a constraints-based model so as to provide flexibility in terms of provider independence as well as giving an abstract view of infrastructure capabilities. We describe a two-phase software abstraction model which facilitates infrastructure resource discovery across multiple providers. An application's hosting requirements are specified by means of constraints. We describe an abstract interface for managing resources in a multi-provider environment and study how infrastructure features and user requirements can be expressed and used to find suitable resources for hosting an application.

Background

An *infrastructure provider* is an entity that offers resources for lease according to a specific pricing model, along with a management interface for users to browse, purchase, monitor and control resources. A resource may be a *compute resource* with a selected operating system, a *storage resource* or a *service* providing predefined functionality. For most providers a *compute resource* is a virtual machine (VM) with a specific hardware configuration and operating system type - it may also contain other software or data required by a user. *Infrastructure providers* offer different hardware configurations with varying pricing models. Each physical infrastructure resource is owned by an *infrastructure provider* and will usually be shared between a number of users. In this section, we compare the resources offered by three popular *infrastructure providers*. We discuss some of the research works related to this field.

What infrastructure providers offer

Currently, it is difficult to compare the different resource options. Comparision of resources is likely to become more complex as additional value-added services are offered by providers directly. Table 1 provides a summary of resources possibilities offered by three different infrastructure providers.

Hardware configuration unit

Amazon EC2 provides hardware configurations using variety of *instance types – Standard*, *Micro*, *High-Memory*, *High-CPU*, *Cluster-Compute*, or *Cluster-GPU* instances; and different *sizes – Small*, *Medium*, *Large*, *Extra Large*,

Table 1 Comparison of product offerings by Amazon EC2, Flexiscale and GoGrid

		Amazon EC2	Flexiscale	GoGrid
Hardware Configuration Unit		*Sizes*: small, medium, large, extra-large, etc; *Instance Types*: standard, micro, high-memory, high-cpu, etc.	*Server Unit*: combination of different CPU, memory and storage.	*Cloud Server Size* based on RAM.
Operating System Type		Amazon Machine Images (AMI).	Standard Windows or Linux ISO images.	GoGrid Server Images.
Pricing Structure	Virtual Machine	On-demand Instance, Reserved Instance and Spot Instance pricing.	Server Units.	Based on the amount of RAM deployed (RAM hours)
	Storage	Storage attached to VM is free; Other storage offerings like Elastic Block Store (EBS) is charged based on provisioned storage and I/O request; Simple Storage Service (S3) is charged based on bucket location, redudancy durability and data transfer request.	Disk space is charged per month per GB of storage space allocated no matter it is attached to a server or not.	Additional storage space is charged by monthly usage.
	Data Transfer	Based on availability zones and regions; No charge within same region.	Traffic on private VLAN is free; inbound and outbound traffic on public VLAN is charged.	Inbound transfer is free; outbound transfer is charged.

The following table compares the product offerings by Amazon EC2, Flexiscale and GoGrid.

Double Extra Large or *Quadruple Extra Large*. For example, an EC2 High-CPU Medium instance provides a 1.7GB memory, 5 *EC2 compute units* (2 virtual cores with 2.5 units each) and 350 GB of local storage, 32-bit platform virtual machine. A Flexiscale virtual machine is marketed using *server units*. A *server unit* is a combination of different numbers of CPU cores, memory and storage–for example, a virtual machine with 2GB memory and 1 CPU core. GoGrid supplies infrastructure using *Cloud Server Size*, based on RAM. A GoGrid cloud server can be a virtual machine with 2 CPU cores, 100GB storage and 2GB RAM.

Operating system type

An infrastructure resource is associated with an operating system type. In Amazon EC2, an operating system is offered as *Amazon Machine Images* (AMIs). Each AMI is pre-configured with a specific operating system type and possibly bundled with different software. A standard Amazon AMI could be a Ubuntu 10.04 Lucid operating system with (or without) pre-installed Java 1.6 and JBoss applications. Flexiscale offers standard Windows (e.g. Windows Server 2008) or Linux (e.g. CentOS Linux 5.4) operating systems. Users can install their own operating systems by downloading and booting from *ISO images*. GoGrid provides operating systems by means of *server images* which can be a standard Windows (e.g. Windows Server 2008) bundled with additional packages such as IIS 7.0, Fast CGI PHP, MS SQL Server Standard 2008 database software or just a bare operating system with no extra packages.

Pricing structure

There is a wide range of pricing structures for infrastructure resources. The provider Amazon EC2 has a variable price structure for using infrastructure resources; the cost of a resource depends on the *size* of the instance and the *region* in which it is running. Prices for using Linux or Windows also vary. As well as standard pricing for *On-Demand Instances*, Amazon EC2 also provides *Reserved Instance* and *Spot Instance* pricing structures. The pricing of *Reserved Instance* is a one-time payment for a specific period which offers a significant discount for the instance running within that period. *Spot Instance* provides a way to bid for and possibly purchase unused Amazon EC2 capacity. The price charged for *Spot Instance* (*Spot Price*) fluctuates with the level of demand, and is typically significantly less expensive than the normal Amazon EC2 pricing. When bidding for a *Spot Instance*, a user specifies a maximum bid price. If a *Spot Price* exceeds a user's maximum bid price, then the VM instance is terminated by Amazon EC2 and no charged is applied for usage. Otherwise the user leases the VM for a period of time and the charge for usage is the *Spot Price* for the usage period. The internet data transfer cost depends on the Amazon EC2 *Availability Zones* and *Regions*. Other than default local VM storage, Amazon EC2 also provides other storage options, namely Elastic Block Store (EBS) and Simple Storage Service (S3). The charge for EBS usage is based on the provisioned storage and I/O requests. The charge for S3 is based on the S3 bucket location, redundancy durability and the data transfer request (PUT, COPY, POST, LIST, GET).

Flexiscale services are priced in terms of *units*. A package of *units* must be purchased in advanced to enable usage of Flexiscale services. *Units* can be used to purchase four types of Flexiscale service, namely *Servers*, *Disk*, *Network* and *Software Images*. Each type of service is priced over a specific time period. The charge for a Flexiscale server depends on the number of CPU cores and the amount of RAM acquired. Disk usage is charged by allocated disk space and the number of I/O accesses made to disk. Inbound and outbound network traffic on the public *VLAN* is also charged. Each account is allocated blocks of public IP addresses and private VLANs to use as they wish. Open source operating systems are free but usage of Windows operating system activates a further charge.

GoGrid pricing depends on server *RAM hours*, outbound data transfers and storage usage. GoGrid provides a pay-as-you-go pricing model as well as monthly pre-paid plans which offer better value for most applications. Only outbound data transfer is charged (inbound data transfer is free). Cloud storage by GoGrid is optional and is determined by the maximum storage space utilisation within a given billing cycle.

Related work

There are a number of alternative approaches for offering and selecting resources for the cloud marketplace. For example, there are a number of open source *IaaS* provider implementations that expose clusters of VM hosts as *IaaS* resources. These include Eucalyptus [6], OpenNebula [7] and Nimbus [8]. Most clone Amazon's EC2 API to allow easy access through existing technologies. These implementations offer a set of cloud management operations; however, they are resource-centric, giving external definitions of the capabilities of particular resource types and prices (refer Table 1).

The RESERVOIR [9] framework creates an environment where different cloud services (or resources) can be managed together. Services are encapsulated in a Virtual Execution Environment (VEE). Virtualisation technology and physical resources are represented using a Virtual Execution Environment Host (VEEH), which is managed by the Virtual Execution Environment Management (VEEM) system. VEEM utilises OpenNebula and manages the deployment and migration of VEE on VEEH.

A Service Manager (SM) is introduced to instantiate the service applications and manage the service level agreement (SLA). The RESERVOIR framework provides a service specification mechanism [10] to define application configurations by extending DMTF's OVF standard [11]. This service specification includes VM details, application settings and deployment setings. The RESERVOIR's approach gives a provider-focused view of resources. It allows service providers to specify a complete definition of their cloud services.

The SLA@SOI [12] framework introduces a multi-layer SLA management framework for service-oriented infrastructures. The SLA@SOI model consists of terms, service level objectives (SLO) and conditional rules. Terms are the attributes that define the infrastructure resources, such as number of CPU cores, memory allocation, or resource's geographical region. SLOs are the attributes for monitoring resource performance, such as CPU utilisation. Conditional rules define the actions that need to be executed if there are changes in resource performance (SLA). The SLA@SOI approach is focused on the management, monitoring and readjustment of SLAs.

The Intercloud project [13] implements an ontology-based resource catalog that captures the features and capabilities offered by cloud providers. The proposed ontology defines the physical attributes of resources, such as CPU and storage. It also defines other features of resources, such as security, recovery, and compliance capabilities from the provider's point of view.

The Mosaic [14,15] project studies the cloud interoperability and portability issues. It also tries to address service discovery, service composition and SLA management issues. It proposes an ontology [16] for resource annotations. The Mosaic's ontology defines a set of non-functional and functional properties which are used to describe cloud resources. The definition of SLA in Mosaic follows the SLA@SOI concepts.

A resource discovery model for multi-Cloud applications

The proposed resource discovery model is based on the notion of service-centric systems which are subsequently deployed as dynamic applications which reside in a multi-provider cloud. The usage of infrastructure resources is application dependent: some applications are implemented using a dynamic collection of infrastructure resources; others only use cloud resources during periods of high load, or to provide disaster recovery.

An infrastructure provider agnostic approach

In the model, applications are agnostic to the underlying provider; this means that applications can use the most appropriate cloud resources when they are needed. The Zeel/i framework[17] uses the customised mechanisms of a host provider to configure an environment for a user. The environment is wrapped in a provider-agnostic API which insulates applications from provider APIs (see Figure 1). The first step in using Zeel/i is to determine the current resource marketplace, i.e. the cloud of resources currently made available from resource providers which satisfy the application's requirements. The choice of the most suitable provider depends on the constraints of the application and also on the kind of financial arrangements that suit the application owner (forming a financial agreement with a provider may be as simple as creating an account and providing credit card details, but it could also involve personally auditing the provider's data centres for particular application domains, such as financial services). Once a pool of resources is defined, an application can select and allocate appropriate resources. In order to be provider-neutral, Zeel/i takes a discovery-based approach where an application attempts to *find* resources (from the pool of acceptable providers) that meet its requirements– this approach is explained in detail in the next section. Once a collection of feasible resources, called *resource*

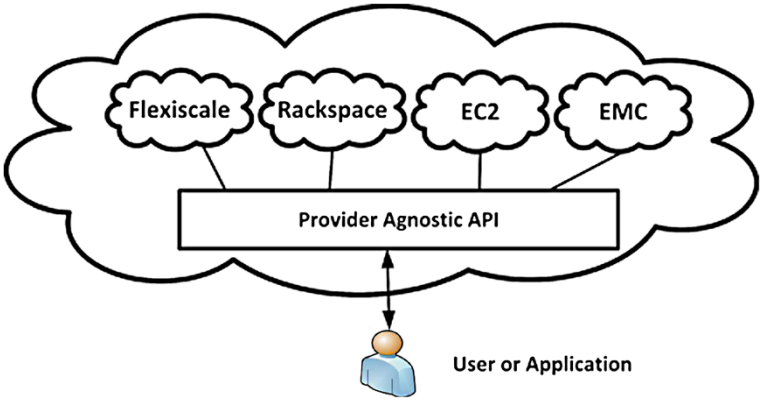

Figure 1 A provider-agnostic resource cloud model. A provider-agnostic API that insulate users or applications from different implementation of different provider API.

templates, have been identified for an application, they can be filtered using a heuristic (e.g. cheapest cost) to select the best match for an application.

Once an appropriate resource has been identified, it can be *reserved*. A reservation is a short-term hold on resources designed to provide a guarantee that the resources will be available when needed. Once resources are needed they are *instantiated*, which provides the application with a running resource which it can use. When the application finishes with a resource, it *discards* it by returning it to the provider. An illustration of the resource discovery and utilisation model is shown in Figure 2. The resource model allows application users to specify resources without the knowledge of the internal implementation of the underlying infrastructure provider. This insulates users from the frequent changes that arise in an underlying provider's APIs (for managing infrastructure resources). The resource model allows applications to take a dynamic, commodity-based approach to resource usage. An application cloud can be deployed and scaled according to application and system constraints, costed within a given budget and have portability over the set of available infrastructure providers.

Using two-phase constraints-based discovery approach

Currently, cloud providers advertise their resources through websites. Typically, users write code targeted for a single cloud provider, statically selecting resource types for their applications. Often they build customised operating system images to enable their software to be deployed onto these resources. When a provider changes its pricing structure or adds new features and value-added services (which they do on a near-weekly basis), users must re-evaluate their cloud environment and decide whether they should switch provider, resources, or even modify their operating system images.

This static human-driven approach does not scale to a world in which multiple providers, each with their own unique features, compete for users' workloads. Providers are interested in packaging and selling infrastructure products–their products include templates for resource types, each with a given amount of RAM, CPU count and often some scratch storage. Typically, the requirements of an application are more complex than available resource templates; for example,

- a database component may require storage resilience of 99.9999%;
- a transcoder used in a digital media video application may *prefer* a CPU with the SSE 4.2 instruction set;
- low latency between some services is required if acceptable performance is to be guaranteed.

Currently, there is a mismatch between the provider perspective of neatly packaged ranges of infrastructure products and the application perspective of constraints between services.

Constraints optimisation engine

In [18], we propose a model to formulate the application requirements into constraints. Various types of constraints are identified:

- a *hardware constraint* refers to hardware requirements of the application, such as CPU or RAM.
- a *storage constraint* refers to storage requirements of the application, such as persistent storage space.
- a *software constraint* refers to any software or service utilised by the application, such as Java JDK 1.6.
- a *performance constraint* refers to the Quality of Service (QoS) or Service Level Agreement (SLA) that the application need to achieve, such as response time and network latency.
- a *cost constraint* refers to the budget or cost restriction of the application, such as a maximum of $1 per hour per infrastructure resource.
- a *compliance constraint* refers to any standard, regulatory or compliance requirements that the application need to comform with, such as data center standard TIA-942 or UK Data Protection Act 1998.

The two-phase constraint model is illustrated in Figure 3. An application's constraints are partitioned into *hard constraints* and *soft constraints*. *Hard constraints* refer to *must-have* requirements which persist and are invariant during execution, such as CPU, memory, or legislation regulations; *soft constraints* refer to desired requirements which can change or be re-prioritised, such as the cost of consuming resources. The classification of requirements into *hard constraints* or *soft constraints* depends on user's need. For example, network latency can be classified as *soft constraints* if an application *prefer* a fast response time; however, one could classify network

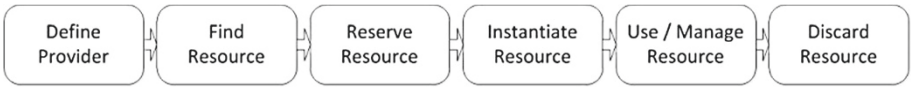

Figure 2 A provider-agnostic resource cloud lifecycle model. The resource cloud lifecyle model.

Figure 3 Two-phase resource selection from providers. A two-phase resource selection approach involving hard constraints and soft constraints.

latency as *hard constraints* if the application's response time must not exceed certain threshold limit.

In the first phase, the proposed model only utilises *hard constraints*. A set of possible resources is generated which satisfies the constraints; each resource is associated with a *cost model* which can be interrogated with *usage scenarios* (such as "4 days of use, 8GB of inbound internet bandwidth, 4GB of outbound internet bandwidth, 1TB of data stored for 4 days") to determine a cost, taking into account of bulk discounts, special offer periods, etc.

The following code fragment illustrates how application constraints can be expressed:

```
HardwareConstraint cpu = new
  CPUCoreConstraint(4)
HardwareConstraint ram = new
  RAMConstraint(8, GIGABYTES)
SoftwareConstraint os = new
  OperatingSystemConstraint
  (UBUNTU_10_04)
StorageConstraint storage = new
  StorageConstraint(1, TERABYTES,
  new StorageResilence(99.9999))
ComplianceConstraint compliance =
  new DatacenterConstraint
 (Compliance.TIA942)
Resource[] resources = FindResources
  ByConstraints(cpu, ram, os,
  storage, compliance)
```

The proposed model uses a *Constraints Optimisation Engine* to find the most appropriate resources for an application. The engine combines data from a *Box Provider*, a *History Database*, a *Specification Factory*, a *Usage Scenario Estimator*, and a *Constraints Validator*. The constraints optimisation process is illustrated in Figure 4.

The *Box Provider* supplies information about those infrastructure providers with which the application user has already established a financial agreement. The model allows a user to choose which providers to activate in

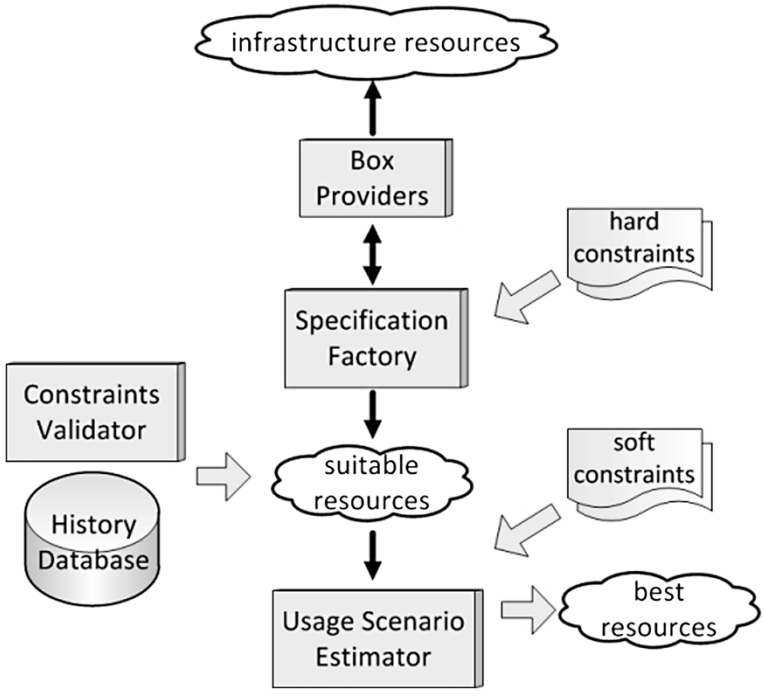

Figure 4 Constraints optimisation process. A constraints optimisation process.

the system. By adopting a provider-agnostic approach, the model offers a single and harmonised view of multi-cloud infrastrasture resources.

The *History Database* is a component that holds historical monitoring data. This includes information such as provisioning latency, probabilities of failure for particular providers, hardware offers, application-specific performance and other general performance benchmarks such as CPU, RAM, Disk, or Network data. Historical monitoring data allows the system to select the most appropriate resource based on actual observed behaviour, rather than provider-advertised performance data. This allows users to react immediately to cloud problems, such as provider failure. Consequently, the provisioning system can avoid resources that have a history of unreliability.

The *Specification Factory* component can combine all of the hardware and software options that are made available by infrastructure providers in order to generate a pool of available resources–*resource templates* (*search space*). For example, a compute resource offered by Amazon AWS EC2 could be a *High-CPU Medium instance* combined with a *Ubuntu 11.04 Natty AMI image* in the *AWS EU region*; or it could be a *Standard Small instance* combined with a *Windows Server 2008 AMI image* in the *AWS US-East region*. In order to allow runtime discovery of resources which satisfies the constraints of the application it is necessary to have a clearly defined description of cloud resources. There are similarities between the *Specification Factory* and intercloud research; for example, in Bernstein et al. [19], cloud resources are described using semantic web techniques. The proposed solution is based on high-level constraints which can be applied to multiple description formats–for example, against a set of Java interfaces, SPARQL queries against an RDF ontology or constraints in a rules engine–whilst keeping an application-based viewpoint to keep the model at a level appropriate to users.

The *Constraints Validator* is used to filter the resource set (*search space*) by ensuring that all application constraints are satisfied. The engine makes a *move* by choosing a resource from the *search space*, and checking if that particular resource meets the application constraints; if the constraints are satisfied, then the *move* is valid. A *solution path* is formed by combining (chaining) each valid *move* (see Figure 5). A set of *solution paths* (*solution space*), which satisfy the application constraints, will be generated. If none of the available resources satisfy the constraints, no solution is generated. A "degraded" *solution path*, which only partially satisfies the application constraints (e.g. with lower hardware specifications) could be offered. The use of "degraded" *solution paths* is not considered in this paper.

The *Usage Scenario Estimator* can be used to estimate the cost of a specific *usage scenario*. The engine generates the *solution space* (identified from the *Constraints Validator*) and assigns a *cost* to each *solution path*. The *cost* is an estimate of a typical *usage scenario* for a particular application, with *chargeable* operations, such as:

- the cost of allocating/deallocating a resource;
- the cost of reserving a resource;
- the cost of keeping a resource switched on for an application-specified period of time;

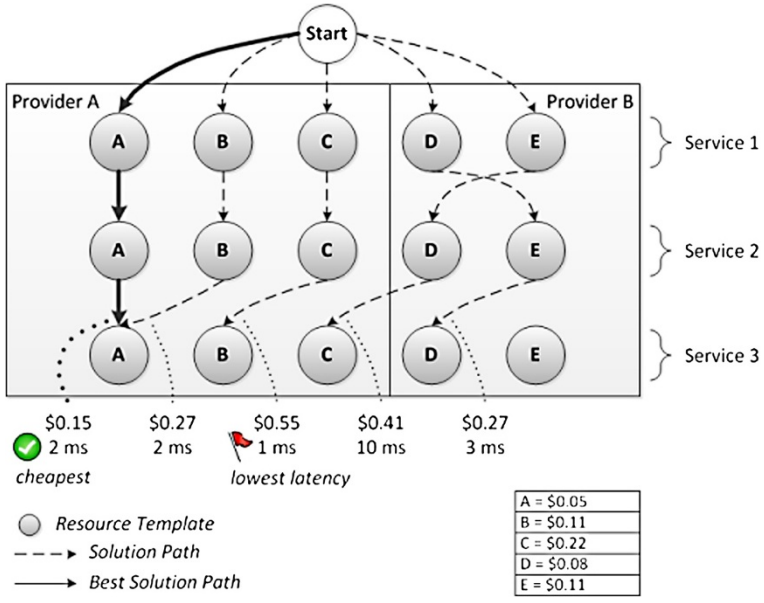

Figure 5 Constraints optimisation solution paths. A solution path using the proposed constraints optimisation mechanism.

- the cost of using a certain amount of incoming/outgoing bandwidth within the provider (for an application-specified period of time);
- the cost of performing a number of I/O operations on a local or a SAN disk (for an application-specified period of time);
- the cost of storing a given volume of data on a local or a SAN disk (for an application-specified period of time).

A heuristic can give weights to each of these factors in order to obtain an application-centric metric. In the proposed model, cloud application developers create costing scenarios as an integral part of the development process– this adds an additional development cost but it simplifies the process of integration in a multi-provider marketplace. The model provides a number of predefined scenarios, such as, compute-only, network-only and storage-only: these scenarios allow developers to use the model without significant extra effort. Once a metric is identified, the engine can select the best *solution path* that match the application requirements.

For example, suppose that a user needs to provision a web application which requires three compute resources with the following requirements:

- Ubuntu Maverick operating system (*hard constraint 1*)
- at least 1 CPU, 1 GB Memory, 20 GB disk space for each compute resources (*hard constraint 2*)
- must be located within EU region (*hard constraint 3*)
- must be from a reliable infrastructure provider with at least 99% uptime (*hard constraint 4*)
- a maximum budget of $0.50 per hour for 30 days period (*soft constraint 1*)
- a network latency of less than 100 milliseconds between each compute resources (*soft constraint 2*)

In the proposed model, the *Box Provider* component establishes a provider-agnostic view of a multi-provider infrastructure cloud. The *Specification Factory* generates a list of *resource templates* which denote the resource offerings from different infrastructure providers. The initial set of *resource templates* could be huge in size, depending on the number of infrastructure providers being used. For example, nearly half a million *resource templates* could be generated for the Amazon AWS provider (by combining different AWS *instance types* with different *AMI images*, *regions* and *availability zones*). The size of the *search space* varies depending on the search algorithm. A naive exhaustive search to find Y candidates from a pool of X resources could generate (X^Y) different combinations (assuming the same type of *resource template* can be reused). In practice, it is highly desirable to limit the *search space* by imposing sufficient *hard constraints* using different heuristic algorithm. For example, if *location* treated as a *hard constraint*, then the *search space* could be significantly reduced (see Table 2). In phase-one of the proposed model, the *Constraints Optimisation Engine* identifies a subset of suitable *resource templates* which meet all of the *hard constraints*. These templates are then validated using information from the *History Database* and *Constraints Validator*. In phase-two search of the model, the engine applies a *cost model* to each feasible *resource template*, allowing the *Usage Scenario Estimator* to compare the *soft constraints*. For a usage scenario where the *cost constraint* is the major factor, the engine assigns a higher weighting to this constraint and tries to identify the cheapest *solution path* (Figure 5) *at the time* that the infrastructure is being deployed. In practise, usage scenarios may change over time. If the *network latency* becomes more important than *cost*, then the engine assigns a higher weighting to the *network latency* (and possibly selects a different *solution path*).

Table 2 Experimental result for financial services application

Application's Requirements **4 CPU, 4GB RAM, 1000GB Disk, 64-bit platform, Ubuntu Maverick** **within EU region** **max $1 per hour**							
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**Total**
No constraints	244552	63282	78972	25668	39108	1248	452830
Hardware / Software (hard constraints)	24	18	18	12	12	9	93
Location (hard constraints)	0	0	18	0	0	9	27
Cost (soft constraints)	0	0	12	0	0	9	21

Infrastructure Providers:

P1 = AWS US East ; P2 = AWS US West ; P3 = AWS EU West

P4 = AWS AP North East ; P5 = AWS AP South East ; P6 = Flexiscale UK

Suitable resource templates available from different providers after applying different constraints.

This model provides a unified framework for comparing infrastructure resources in order to select the most appropriate combination of resources for a given application. In the model, applications are insulated from the rapidly changing provider APIs and from the dynamically changing collection of infrastructure products and services. The view of the model is *application oriented* and infrastructure products are chosen to satisfy the application requirements.

Exemplar applications

A financial services application and a high performance computing application are used to illustrate the execution of the proposed resource discovery mechanism.

A financial services application

The financial services domain provides a challenging environment for infrastructure deployment. Other than general requirements, such as hardware (CPU, memory), software (C++ compiler) and storage (disk space), the financial services domain requires the use of other stringent requirements, such as data quality and integrity, security (encryption and authorisation), performance (response time, network latency) and compliance (legislation regulations) [20]. In order to satisfy the constraints of high availability and high site resilience, multiple mirror-infrastructures can be provisioned in different geographical locations (location constraints); however, the performance constraints may require low network latency between different sites. These *opposing* requirements of geographical separation and low latency have the potential to severely prune the solution space.

We have worked with a financial services company to develop a prototype system to provision a resilient financial infrastructure. The simplified infrastructure in Figure 6 shows a *financial services feed* which streams data to two *database replicas*, against which worker jobs, in a dynamically deployed *Condor[21] pool*, run queries and perform processing. The following constraints are used to guide infrastructure product selection.

The data coming from the financial services feed has an approximate flow rate of 500Kbps from 8.00am to 4.30pm (approximately 2 gigabytes per day). This information is sufficient to model the bandwidth cost involved in a potential database replica placement. The database replicas require:

- incoming internet bandwidth of at least 500Kbps from the financial services feed and outgoing 500Kbps to the other database replica
- between 10Mbps and 20Mbps of bandwidth to the condor pool nodes (to satisfy each node's 2Mbps requirement)
- minimal latency to ensure that the condor pool acquires timely data feeds despite synchronisation
- sufficient redundancy and site resilience to ensure that there is a very high probability that at least one infrastructure branch survives a major provider failure
- legislation compliance to ensure that no data is stored or transmitted outside of the EU region.

The condor pool should have the lowest possible latency link to the database server. They do not have to be geographically distributed. This latency requirement, along with the condor pool bandwidth constraint results in a pool being placed on the same provider's resources. A key benefit of our proposed approach is its two-phase nature as illustrated in Figure 3.

The initial search phase identifies only resources which are *suitable* for the Database Replica Nodes and the Condor Nodes. The *hard constraints* include the hardware specifications and advertised/observed reliability, as well as the software configuration, such as the operating system type associated with it. The *hard constraints*

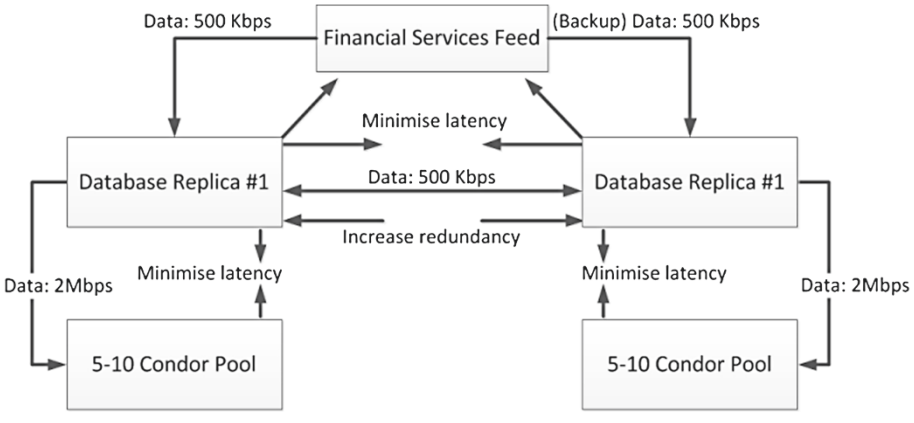

Figure 6 An example financial services setup. A simplified infrastruture for a financial services feed application.

also include location restriction. For example, as part of the financial services compliance requirements, no data should be hosted or transmitted outside the EU region.

The second phase selection balances the complexities of cost, non-functional requirements and preferences across many providers, known as *soft constraints* in our proposed model. Delaying consideration of cost until the second phase also enables a much more sophisticated view of the price of a system as a whole rather than the cost of individual resources from providers–considering costs per resource may not result in the most appropriate infrastructure for an application. The infrastructure for the financial system is shown in Figure 6; the compute nodes to be used for Condor are expensive while the database replica server is relatively cheap. This results in an application infrastructure with low cost. A search based solely on individual resources could reject the condor segment of the infrastructure as being too expensive.

A high performance computing application

Conventional high performance computing (HPC) applications run on super-computers, compute clusters or grids, which typically reside within organisation boundaries. Cloud Computing offers an alternative platform for executing HPC application.

Block Matrix Multiplication is used as an example of how HPC applications can be mapped onto Cloud infrastructure resources. The multiplication of an $m \times p$ block matrix A and a $p \times n$ block matrix B can be defined as:

$$1 \leq i \leq m \,.\, 1 \leq j \leq n \,.\, C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

where each element of matrices A and B is defined as a sub-matrix. The following example illustrates the cases where $m, n, p = 2$ (i.e. 2×2 block matrices).

Consider two $16K \times 16K$ matrices, each composed of four $8K \times 8K$ square blocks. A straightforward multiplication requires 8 matrix-matrix multiplications followed by four matrix additions. The operations can be carried out in parallel using 8 Cloud compute resources. Each matrix-matrix multiplication requires three matrices (two input matrices and one result matrix) in memory during execution. Using 64-bit precision, each $8K \times 8K$ matrix requires 512 MB RAM. Therefore each Cloud compute resource must have at least 1.5 GB RAM. Another possible constraint is fast turnaround-time. In this case reliable high performance compute resources are preferred.

Experimental result for resource selection

A prototype demonstrator has been implemented using the proposed two-phase resource discovery model. Some infrastructure providers offer resources across multiple geographical regions. For example, in the demonstrator, Amazon AWS is considered to be 5 sub-providers in different regions–US east, US west, EU west, AP north east and AP south east. Infrastructure resources are described in terms of *resource templates*, annotated with appropriate metadata. Metadata information includes hardware specification (e.g CPU, RAM, storage capacity), operating system type, resource location and performance reliability.

Resource selection result for financial services application

In the financial services experiment, Amazon AWS and Flexiscale are chosen as the infrastructure providers. *Resource templates* and *pricing structures* are generated dynamically during runtime depending on providers' offerings (e.g. a new *AWS AMI image* is created). Initially, before any constraints are imposed, there are nearly half a million *resources templates* that are made available by different providers. The number of suitable *resource templates* is significantly reduced once *hard constraints* and *soft constraints* are taken into account (see Table 2). In the experiment, the demonstrator takes an average of 32 seconds to execute the resource discovery process. However, almost 95% of the execution time is spent in establishing a connection to the infrastructure providers' API across the internet and populating the *resource templates* in real-time. This execution time could be improved by regularly caching the resource templates in a local database.

Figure 7 shows the experiment result of the financial services feed example. The prototype demonstrator selects one infrastructure site at Amazon Dublin using a *m1.large instance* for the Database and *c1.xlarge instance* for the Condor Nodes; and another infrastructure site at Flexiscale London using *4 CPU 8GB RAM* for the Database and *6 CPU 6GB RAM* for the Condor Nodes. This fulfils the general requirements (e.g. hardware, software, etc) as well as other performance and compliance requirements that each site must be reliable and geographically separated from the other (while maintaining a low-latency link to the data feed and each other).

Resource selection result for HPC application

In the HPC experiment, 90 *resource templates* are found by using the proposed resource discovery model to search for a 64-bit machine with at least 1.5 GB RAM and a Ubuntu Oneiric operating system across all of the available infrastructure providers. Table 3 shows the top 10 cheapest *resource templates*, with the cheapest being an AWS *resource template*, m1.large–64-bit with 2 CPU cores and 7 GB RAM. By provisioning a cluster of machines of the same type, the cheapest cost per hour for 8 compute resources is $2.80. If cost is of higher priority than turnaround-time, then AWS *spot instances* should be preferred. In the proposed model, AWS *spot instances* are differentiated by *resource reliability* attribute. *Spot instances* are annonated as potentially *unreliable* resource

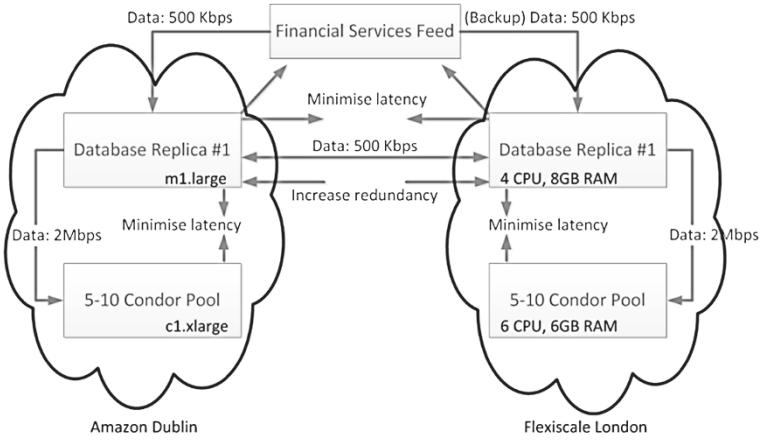

Figure 7 Two-phase resource selection result. The selection result for the financial services infrastructure using the proposed two-phase approach.

type since they may be terminated by the provider if demand for the *spot instances* is strong. Table 4 shows the top 10 of 90 results found from a similar search to that in Table 3, but for *spot instances* instead of normal resources. The cheapest instance is again an AWS *resource template*, m1.large, but the cheapest cost of an 8 compute resources is $0.96 per hour, a saving of $1.84 per hour.

If cost and turnaround-time are both high priorities, then a different candidate could be considered. One way of ensuring faster execution time is to utilize multiple CPU cores on a compute resource. A multi-core compute resource is typically more expensive, but by taking the advantage of AWS *spot instance*, a more affordable compute resource may be identified. Table 5 shows the top 10 of 29 results found in a search for a 64-bit machine with at least 1.5 GB RAM, Ubuntu Oneiric OS and at least 8 CPU cores. The cheapest cost per hour for an 8 compute resource is $1.76. A cluster of 8 c1.xlarge compute resources is provisioned. A Java based parallel Block Matrix Multiplication orchestration has been executed on the cluster. The orchestration is dynamically installed using a multi-threaded optimised GotoBLAS2 [22] library on each machine instance. Table 6 shows the average, minimum and maximum execution times of 20 executions for both $8K \times 8K$ and $16K \times 16K$ block matrix multiplications. For comparison, Table 6 also shows the execution times for computing the multiplication using a single c1.xlarge instance.

The benefits of using two-phase approach

The proposed two-phase approach permits a second phase of selection to be focused on the needs of the application *at the time* that the infrastructure is being deployed. For example, for media applications [23], deployment can be batch based or on-demand, depending on the context in which it is to be used. The application requirements are the same in both cases; however, a batch system does not require high network bandwidth and puts more emphasis on storage size and speed. For the financial services example, the demonstrator could select an alternative infrastructure provider if the cost of provisioning such infrastructure is cheaper *at the time* than the infrastructure in the current one. For example, if the Amazon AWS data transfer price is reduced, then the tool may give an alternative resource solution. In the financial services domain, market conditions may alter the priority of an application's constraints: for example, after a major incident, such as Japan's tsunami or a major downgrade of US credit ratings, the financial analyst may need to perform computation-intensive analysis rapidly. Under these circumstances, the cost of provisioning the infrastructure might become an insignificant constraint; low latency between resources

Table 3 Experimental result for HPC application (normal resources)

Provider	Location	Instance type	CPU cores	RAM (GB)	Cost per hour ($)
AWS US East	us-east-1a	m1.large	2	7	0.35
AWS US East	us-east-1b	m1.large	2	7	0.35
AWS US East	us-east-1c	m1.large	2	7	0.35
AWS US East	us-east-1d	m1.large	2	7	0.35
AWS US East	us-east-1e	m1.large	2	7	0.35
AWS EU West	eu-west-1a	m1.large	2	7	0.39
AWS EU West	eu-west-1b	m1.large	2	7	0.39
AWS EU West	eu-west-1c	m1.large	2	7	0.39
AWS US West	us-west-1a	m1.large	2	7	0.39
AWS US West	us-west-1b	m1.large	2	7	0.39

Top 10 resource templates found for a 64-bit normal compute resource with at least 1.5 GB RAM and a Ubuntu Oneiric OS.

Table 4 Experimental result 1 for HPC application (AWS spot instances)

Provider	Location	Instance type	CPU cores	RAM (GB)	Cost per hour ($)
AWS US East	us-east-1a	m1.large	2	7	0.12
AWS US East	us-east-1b	m1.large	2	7	0.12
AWS US East	us-east-1c	m1.large	2	7	0.12
AWS US East	us-east-1d	m1.large	2	7	0.12
AWS US East	us-east-1e	m1.large	2	7	0.12
AWS EU West	eu-west-1a	m1.large	2	7	0.15
AWS EU West	eu-west-1b	m1.large	2	7	0.15
AWS EU West	eu-west-1c	m1.large	2	7	0.15
AWS AP South East	ap-southeast-1a	m1.large	2	7	0.15
AWS AP South East	ap-southeast-1b	m1.large	2	7	0.15

Top 10 resource templates found for a 64-bit AWS spot instances with at least 1.5 GB RAM and a Ubuntu Oneiric OS.

might be the dominant constraint rather than geographical separation. An alternative solution which completely replicates the infrastructure with a much better performance (but more expensive) is preferred. For the HPC example, a user may alter the priority of the cost constraint or turnaround-time constraint. The demonstrator could select an effective compute cluster which balances the cost and turnaround-time requirements. Our proposed model provides an holistic view of resource discovery which allows us to constrain the cost for an entire infrastructure, infrastructure branch or resource.

Conclusion

Cloud infrastructure providers offer highly flexible and cost effective resources for use by a new generation of network-centric infrastructure applications. The infrastructure marketplace is developing rapidly with new providers, infrastructure products and value-added services coming to the market. This rapid development environment places significant strain on existing infrastructure application users because of the complexity of selecting appropriate resources from a dynamic marketplace.

Mapping an application's requirements onto a set of resources is challenging. We have developed a two-phase resource selection model using a constraints-based approach which enables users to match their applications' requirements to infrastructure resources. The model provides an application-focused, rather than a provider-focused, view of resources. This enables application requirements to be expressed in a domain specific way rather than using the terms used by particular providers. By adopting this application-centric approach, constraints are used to express the requirements of an

Table 5 Experimental result 2 for HPC application (AWS spot instances)

Provider	Location	Instance type	CPU cores	RAM (GB)	Cost per hour ($)
AWS US East	us-east-1a	c1.xlarge	8	7	0.22
AWS US East	us-east-1b	c1.xlarge	8	7	0.22
AWS US East	us-east-1c	c1.xlarge	8	7	0.22
AWS US East	us-east-1d	c1.xlarge	8	7	0.22
AWS US East	us-east-1e	c1.xlarge	8	7	0.22
AWS EU West	eu-west-1a	c1.xlarge	8	7	0.30
AWS EU West	eu-west-1b	c1.xlarge	8	7	0.30
AWS EU West	eu-west-1c	c1.xlarge	8	7	0.30
AWS US West	us-west-1a	c1.xlarge	8	7	0.30
AWS US West	us-west-1b	c1.xlarge	8	7	0.30

Top 10 resource templates found for a 64-bit AWS spot instance with at least 1.5 GB RAM, 8 CPU cores and a Ubuntu Oneiric OS.

Table 6 Execution time for Block Matrix Multiplication

Matrix size	Block size	Instances used	Avg	S/D	Min	Max
8000	8000	1	41	23.77	24	76
	4000	2	34.8	11.75	26	77
	4000	4	18.25	3.02	15	26
	4000	8	18	5.14	15	36
16000	16000	1	297.22	55.81	171	408
	8000	2	270.95	35.25	246	365
	8000	4	151.95	20.75	94	206
	8000	8	103.55	22.73	77	155

Average, Standard Deviation, Minimum and Maximum times (in seconds) for Block Matrix Multiplication using AWS c1.xlarge instances.

application–sets of such constraints determine the multi-resource requirements of an application. This approach is usually applied in a two-phase manner that enables users to select appropriate resources and then balance the needs of the application infrastructure with functional and non-functional requirements.

The constraints optimisation engine proposed in our model is still in preliminary stage. We are currently improving the engine by considering other optimisation techniques [24-26] and rules engines [27,28]. We are also investigating the possibility of describing application requirements using domain specific ontologies. Scalability is likely to be a problem when a large number of infrastructure providers are considered. We are investigating a three-phase (or multi-phase) approach, which incorporate an additional phase to filter suitable providers before searching for resources. We believe that our approach offers an effective mechanism to compare and select resources from the myriad of providers and infrastructure products.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
The investigation of a resource discovery model for multi-provider cloud was a joint research involving all of the authors. TH and RP initially defined the research theme. TH, PW, YLS designed and implemented the model, performed the financial services experiment and analysed the results. AK and AS performed the HPC experiment and analysed the results. TH and AS supervised the research. All authors read and approved the final manuscript.

Author's information
PW is a Software Engineer at Belfast e-Science group. He received a BEng in Computer Science from Queen's University Belfast in 2005. YLS is a PhD student in Computer Science at Queen's University Belfast. He received a MTech in Software Engineering from National University of Singapore in 2005. TH is the Technical Director of the Belfast e-Science group. He received a PhD in Computer Science from Queen's University Belfast in 1987. AK is a PhD student in Computer Science at Queen's University Belfast. He received a MEng in Computer Science from Queen's University Belfast in 2006. AS is a Lecturer at Queen's University Belfast. He received a PhD in Computer Science from Queen's University Belfast in 1986. RP is Emeritus Professor of Software Engineering at Queen's University Belfast, Director of the Belfast e-Science group, and Visiting Professor at University of Oxford. He received a PhD in mathematics from Queen's University Belfast in 1969.

Acknowledgements
This work is supported by the UK Technology Strategy Board under grant TP/3/PIT/6/I/15656, the UK EPSRC under Platform grant EP/F066139/1 and ECHO grant EP/I03405X/1, and the Knowledge Transfer Secondment (KTS) award KTS-11-12.

Received: 10 February 2012 Accepted: 16 May 2012
Published: 21 June 2012

References
1. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2. Accessed 04 Jan 2012
2. ElasticHosts. http://www.elastichosts.com/. Accessed 06 Jan 2012
3. GoGrid. http://www.gogrid.com/. Accessed 04 Jan 2012
4. FlexiScale. http://www.flexiant.com/products/flexiscale/. Accessed 04 Jan 2012
5. RackSpace. http://www.rackspace.com/. Accessed 06 Jan 2012
6. Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D (2009) The Eucalyptus Open-Source Cloud-Computing System. In CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International, Symposium on Cluster Computing and the Grid 124–131. Washington, DC, USA: IEEE Computer Society
7. Montero RS (2008) OpenNebula: The Open Source Virtual Machine Manager for Cluster Computing. In Open Source Grid and Cluster Conference. Oakland, CA
8. Marshall P, Keahey K, Freeman T (2010) Elastic Site: Using Clouds to Elastically Extend Site Resources. In 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing 43–52. IEEE
9. Rochwerger B, Breitgand D, Levy E, Galis A, Nagin K, Llorente IM, Montero R, Wolfsthal Y, Elmroth E, Caceres J, Ben-Yehuda M, Emmerich W, Galan F (2009) The Reservoir model and architecture for open federated cloud computing. IBM Journal of Research and Development 53(4): 4:1–4:11
10. Galán F, Sampaio A, Rodero-Merino L, Loy I, Gil V, Vaquero LM (2009) Service specification in cloud environments based on extensions to open standards. In Proceedings of the, Fourth International ICST Conference on Communication System software and middleware, COMSWARE '09 19:1–19:12. New York, NY, USA: ACM, http://doi.acm.org/10.1145/1621890.1621915
11. Open Virtualization Format (OVF) Specification. DSP0243 1.0.0. Distributed Management Task Force. Feb 2009. http://www.dmtf.org/standards/ovf; Access date: 2012-01-09
12. Theilmann W, Yahyapour R, Butler J (2008) Multi-level SLA Management for, Service-Oriented Infrastructures. In Towards a Service-Based Internet, Volume 5377 of Lecture Notes in Computer Science, ed. Mähönen P, Pohl K, Priol T 324–335. Springer Berlin / Heidelberg
13. Bernstein D, Vij D (2010) Using Semantic Web Ontology for Intercloud Directories and Exchanges. In International Conference on Internet Computing 18–24
14. Di Martino B, Petcu D, Cossu R, Goncalves P, Máhr T, Loichate M (2011) Building a Mosaic of Clouds. In Euro-Par 2010 Parallel Processing, Workshops, Volume 6586 of Lecture Notes in Computer Science, ed. Guarracino M, Vivien F, Träff J, Cannatoro M, Danelutto M, Hast A, Perla F, Knüpfer A, Di Martino B, Alexander M 571–578. Springer Berlin / Heidelberg, http://dx.doi.org/10.1007/978-3-642-21878-1_70
15. Petcu D, Craciun C, Rak M (2011) Towards a cross-platform cloud API. Components for Cloud Federation. In 1st International Conference on Cloud Computing & Services Science 166–169
16. Moscato F, Aversa R, Di Martino B, Fortis T, Munteanu V (2011) An analysis of mOSAIC ontology for Cloud resources annotation. In Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on 973–980
17. Harmer T, Wright P, Cunningham C, Hawkins J, Perrott R (2010) An application-centric model for cloud management. In Proceedings of the 2010 IEEE 6th World, Congress on Services 439–446. IEEE
18. Sun YL, Harmer T, Stewart A, Wright P (2011) Mapping Application Requirements to Cloud Resources. In Proceedings of the Euro-Par 2011 Parallel Processing Workshops
19. Bernstein D, Vij D (2010) Intercloud Directory and Exchange Protocol Detail using XMPP and RDF. IEEE Services 2010
20. Sun YL, Perrott R, Harmer T, Cunningham C, Wright P (2010) An SLA Focused Financial Services Infrastructure. In Proceedings of the 1st International Conference on Cloud Computing Virtualization. Singapore
21. Thain D, Tannenbaum T, Livny M (2002) Condor and the Grid. In Grid Computing: Making the Global Infrastructure a Reality, ed. Berman F, Fox G, Hey T. John Wiley & Sons Inc
22. GotoBLAS. http://www.tacc.utexas.edu/tacc-projects/gotoblas2. Accessed 09 Jan 2012
23. Perrott R, Harmer T, Lewis R (2008) e-Science Infrastructure for Digital Media Broadcasting. Computer 41(11): 67–72
24. Gregory J, Lin C (1996). Constrained Optimization In The Calculus Of Variations and Optimal Control Theory. London: Chapman & Hall
25. Yeoh W, Felner A, Koenig S (2009) IDB-ADOPT: A Depth-First Search DCOP Algorithm 132–146. Berlin, Heidelberg: Springer-Verlag, http://dl.acm.org/citation.cfm?id=1614611.1614620
26. Bistarelli S, Foley S, O'Sullivan B, Santini F (2009) From Marriages to, Coalitions: A Soft CSP Approach. In Recent Advances in Constraints, Volume 5655 of Lecture Notes in Computer Science, ed. Oddi A, Fages F,

Rossi F 1–15. Springer Berlin / Heidelberg, http://dx.doi.org/10.1007/978-3-642-03251-6_1

27. Fages F, Martin J (2008) From rules to constraint programs with the Rules2CP modelling language. In In Recent Advances in Constraints, Revised Selected Papers of the 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2008, Lecture Notes in Artificial Intelligence 66–83. Springer-Verlag
28. JBoss Drools Planner. http://www.jboss.org/drools/drools-planner. Accessed 10 Jan 2012

doi:10.1186/2192-113X-1-6
Cite this article as: Wright *et al.*: A constraints-based resource discovery model for multi-provider cloud environments. *Journal of Cloud Computing: Advances, Systems and Applications* 2012 **1**:6.